CIG Engineering Pitch — “Hybrid Client-Side Damage Simulation with Server-Side Validation”

This proposal outlines a practical, incrementally adoptable approach to offloading interior damage simulation to the client while maintaining full server authority over combat outcomes. The goal is to reduce server load, improve interior stability, and strengthen anti-cheat integrity without requiring a fundamental rewrite of existing systems.

Overview

When a player enters a ship (either by selection or boarding), the server transmits the complete ship dataset — component definitions, room layout, armor profiles, resistances, and current damage state. This ensures the client has the same deterministic data the server uses for validation.

To minimize RAM and SSD usage, this dataset is streamed using the same distance-based logic already employed by Object Container Streaming. Large ships only load when the player is close enough to interact with them, while small ships can be streamed simultaneously with negligible overhead. Upon entering a ship, only that ship retains its full dataset; all others revert to lightweight exterior state. This keeps the memory footprint low while guaranteeing deterministic simulation for the ship the player is actively using.

During gameplay, the server continues to authoritatively determine hit events: which component was struck, the damage type, impact vector, and the expected damage envelope for that hit. The client then performs the interior simulation locally, generating heat, pressure changes, oxygen loss, structural stress, and secondary effects based on the systemic rules already defined in the ship data.

The client returns a compact damage result vector to the server. The server validates this vector using deterministic range checks, correlation rules, and impossible-state detection. Any deviation outside the expected envelope is corrected server-side, and repeated anomalies are flagged for anti-cheat profiling.

Why This Is Implementable

- **No new simulation systems are required.** The client already runs interior physics, room systems, and component logic for local prediction. This proposal simply formalizes and validates that output.
- **Server authority is preserved.** The server still determines hits, expected ranges, and final outcomes. The client cannot invent damage; it can only simulate within server-defined constraints.
- **Data already exists.** Ship definitions, component stats, resistances, and secondary effect profiles are authored in DataForge. These can be packaged and transmitted on boarding with minimal overhead.

- **Validation is lightweight.** Range checks and correlation rules (e.g., "≥2 subsystems returning zero is impossible") are computationally trivial and scale linearly with the number of affected components.
- **Incremental rollout is possible.** The system can initially validate only HP deltas, then gradually expand to heat, pressure, oxygen, and secondary effects as confidence grows.
- **Client-side simulation does not expose sensitive logic.** The client operates solely on deterministic ship data provided by the server, without access to authoritative combat logic or hidden formulas. All results remain fully verifiable by the server.

Expected Benefits

- **Reduced server load** Interior damage reactions, room system propagation, and secondary effects no longer need to be simulated server-side.
- **Improved interior stability** With the client simulating interior physics locally using authoritative ship data, clipping, rubber-banding, and "falling out of the ship" events are significantly reduced.
- **Stronger anti-cheat** Deterministic validation, impossible-state detection, and server-side correction make subtle manipulation (e.g., "always minimum damage") far riskier and easier to detect.
- **Better scalability** As ships become more complex and multi-crew gameplay expands, offloading interior simulation becomes increasingly important for server performance.

Conclusion

This hybrid model leverages existing systems, preserves server authority, and provides a clear path toward more scalable, stable, and cheat-resistant interior simulation. It aligns with Star Citizen's systemic design philosophy while offering a realistic, engineering-friendly implementation strategy that can be adopted incrementally.

TL;DR

A hybrid damage model where:

- **Server streams only the relevant ship dataset** (distance-based, low memory footprint)
- **Server detects hits and defines the expected damage envelope**
- **Client performs deterministic interior damage simulation** (heat, pressure, oxygen, stress, secondary effects)
- **Client sends a compact DamageResultVector back to the server**
- **Server validates ranges, correlations, and impossible states** (e.g., ≥2 subsystems returning zero)
- **Invalid results are corrected server-side and flagged for anti-cheat**
- **Interior physics become stable** (no more clipping, rubber-banding, or falling through ships)
- **Server load drops significantly** because interior simulation moves client-side while authority remains server-side

Ship Data Synchronization (Within a defined proximity threshold / On Enter / On Boarding)

When a player enters a ship (either by physically boarding it or by selecting it at a terminal), the server transmits the complete ship-specific dataset to the client. This ensures that the client has all required information to perform local interior simulation and damage processing.

Distance-Based Streaming (Memory & SSD Optimization)

To minimize RAM and SSD usage, the ship-data package is only streamed to the client when the player is within a defined proximity threshold. This follows the same logic as existing object-container streaming.

Behavior:

- If multiple ships are nearby, only the closest relevant ship(s) load full interior data.
- When the player walks toward another ship, its dataset streams in while previously loaded ships can be safely unloaded.
- Upon entering a ship, only that ship retains its full dataset; others revert to lightweight exterior state.
- This ensures deterministic simulation without loading unnecessary ships into memory.

This mechanism keeps the memory footprint minimal while still guaranteeing that the client always has the correct authoritative dataset for the ship they are interacting with.

Triggered on:

- Ship selection at a terminal (before spawning)
- Boarding an already spawned ship
- Re-entering after leaving the ship's physics grid

Data transmitted from server to client:

ShipDefinition

- shipTypeId
- componentList
- armorProfile
- roomLayout (rooms, volumes, connections)

ComponentDefinitions

- baseHP
- resistances
- secondaryEffectProfile
- damageValidationProfileId

Current ship state

- componentHP
- damaged/destroyed components
- active fires
- oxygen/pressure levels per room
- structural stress
- power grid status

Purpose:

- Allows the client to run full interior damage simulation locally.
- Ensures the client has identical data to the server for deterministic validation.
- Reduces server load by offloading interior physics and secondary effects.
- Prevents desync because the client always starts with the authoritative state.

Notes:

- This dataset is static per ship type and only changes when the ship is repaired, modified, or damaged.
- The server remains authoritative; the client only simulates and reports results.
- Any deviation from expected ranges is caught by the validation pipeline.
- Distance-based streaming ensures only relevant ships occupy memory, improving performance and scalability.

1. Goals and Principles

Goal of the architecture:

- The **server remains authoritative** over: hits, global state, and final outcomes.
- The **client simulates as much as possible**: interior damage, component reactions, room states, etc.
- **Anti-cheat** is enforced through:
    - server-defined ranges
    - correlation checks (your "≥2 subsystems == 0 → WTF")
    - behavioral patterns (flagged players)

Core principles:

Server:

- decides that a hit occurred
- decides where it occurred (component / position / vector)
- defines the expected range for that hit

Client:

- simulates the full damage chain
- sends back a compact damage vector

Server:

- validates the vector
- corrects it if necessary
- flags suspicious players

2. Data Model (Ship & Damage)

2.1 Ship and Component Data

ShipDefinition

- `shipTypeId`
- `componentList` (list of `ComponentDefinitionRef`)
- `armorProfile`
- `roomLayout` (rooms, volumes, links)

ComponentDefinition

- `componentId`
- `type` (shield, power, thruster, avionics, etc.)
- `baseHP`
- `resistances` (per damageType)
- `secondaryEffectProfile` (heat, fireChance, oxygenLeak, pressureDrop, stress, etc.)
- `damageValidationProfileId`

DamageValidationProfile

- `damageTypeId`
- `hpRangeFunc(hitContext)`
- `heatRangeFunc(hitContext)`
- `fireChanceRangeFunc(hitContext)`
- `oxygenDropRangeFunc(hitContext)`
- `pressureDropRangeFunc(hitContext)`
- `stressRangeFunc(hitContext)`
- `impossibleCombinations` (e.g. `[heat=0 & fireChance=0]`, `[oxygenDrop=0 & pressureDrop=0]`, etc.)

This is your “ship-based component dataset for damage checking”.

3. Client-Side Architecture

3.1 Modules

ClientDamageSimulator

Input:

`HitEvent` from server:

- `hitPosition`
- `hitNormal`
- `damageAmount` (or scale)
- `damageType`
- `componentId` (or resolved from hitPosition)
- `hitContextId` (for range lookup)

Uses locally:

- ship layout + component data (downloaded on boarding)
- current damage state (component HP, room states, etc.)

Output:

- `DamageResultVector` per affected component/room

DamageStateManager (client)

Maintains local state:

- componentHP
- heat
- fire flags
- oxygen
- pressure
- structuralStress
- roomTemp
- etc.

Executes:

- direct effects (impact)
- secondary effects (fire, venting, etc.)
- chained effects (power loss, thruster failures, etc.)

ClientNetEncoder

Encodes the `DamageResultVector` compactly:

Per component:

- componentId (or index)
- bit/byte fields per state:
    - hpDelta
    - flags (fire, offline, destroyed)
    - small diffs (oxygen, pressure, heat, etc.)

3.2 DamageResultVector Structure

Per hit / batch:

```
DamageResultVector {
  shipId
  hitId
  componentResults: [
    {
      componentId,
      hpDelta,
      heatDelta,
      fireChance,
      oxygenDelta,
      pressureDelta,
      stressDelta,
      flags (fireOn, offline, destroyed, etc.)
    },
    ...
  ],
  roomResults: [
    {
      roomId,
      oxygenDelta,
      pressureDelta,
      tempDelta,
      fireFlag
    },
    ...
  ]
}
```

4. Server-Side Architecture

4.1 Modules

HitResolutionSystem (server)

Determines:

- whether a hit occurred (raycast / projectile)
- which component / hull zone was hit
- hitPosition, hitNormal, damageType, baseDamage

Generates:

HitContext:

- shipType
- componentId
- damageType
- impactVector
- gravityContext

ExpectedDamageProfile:

- ranges per subsystem (hp, heat, O_2, pressure, stress, etc.)
- randomized min/max per hit/session

DamageValidationSystem (server)

Receives `DamageResultVector` from client.

For each `componentResult`:

- Lookup `DamageValidationProfile` (ship + component + damageType)
- Generate `ExpectedRange` using `hitContext`
- Perform checks:
  - range validation (hpDelta, heat, etc.)
  - correlation rules (your "≥2 subsystems == 0 → WTF")
  - impossible combinations

Output:

- `ValidationStatus`: OK / SoftFlag / HardFlag
- `CorrectedResult` (if needed)

DamageStateManager (server)

Maintains authoritative state:

- componentHP
- fire
- oxygen
- pressure
- etc.

Applies:

- `CorrectedResult`
- or, if OK: the client-provided result

AntiCheatProfiler

Tracks per player:

- number of SoftFlags / HardFlags
- pattern "consistently near minimum range"
- frequency of suspicious vectors

Actions:

- increased monitoring

- shadow penalties (more server-side checks)
- logs for review
- eventual BAN / restricted play

5. Network Protocol (High Level)

5.1 Server → Client

HitNotification

```
{
  hitId,
  shipId,
  componentId,
  hitPosition,
  hitNormal,
  damageType,
  baseDamage,
  hitContextId (optional),
  serverSeed (for randomisation / obfuscation)
}
```

5.2 Client → Server

`DamageResultVector` (compact encoded, per component/room)

Server decodes → forwards to `DamageValidationSystem`.

6. Validation Logic (Anti-Cheat Layer)

6.1 Range Check

Per field:

```
for each componentResult in DamageResultVector:
    expected = getExpectedRanges(shipType, componentId, damageType,
hitContext)

    if not inRange(componentResult.hpDelta, expected.hpRange):
        flag += 1

    if not inRange(componentResult.heatDelta, expected.heatRange):
        flag += 1

    ...
```

6.2 "≥ 2 subsystems == 0 → WTF" (your rule)

```
zeroCount = 0
if componentResult.heatDelta == 0: zeroCount++
```

```
if componentResult.stressDelta == 0: zeroCount++
if componentResult.oxygenDelta == 0: zeroCount++
if componentResult.pressureDelta == 0: zeroCount++
if componentResult.fireChance == 0: zeroCount++

if zeroCount >= 2:
    hardFlag++
```

Refinements:

- `>= 3` → instant HardFlag
- `== 2` → SoftFlag + profile logging

6.3 Impossible Combinations

Examples:

```
if componentResult.fireChance == 0 AND componentResult.heatDelta == 0:
    hardFlag++

if roomResult.oxygenDelta == 0 AND roomResult.pressureDelta == 0
   AND expectedRoomLeakProbability > threshold:
    softFlag++
```

6.4 Server Correction ("force set")

For HardFlag or clear deviation:

```
if hardFlag > 0:
    corrected = clampToExpected(componentResult, expectedRanges,
maybeMultiply=2.0)
    apply(corrected)
else:
    apply(componentResult)
```

7. Flow: From Hit to Final State

Server:

- Detects hit → `HitResolutionSystem`.

Server:

- Determines component + `HitContext`.
- Generates `ExpectedDamageProfile` (ranges + randomized min/max).
- Sends `HitNotification` to client.

Client:

- Receives hit.
- `ClientDamageSimulator` simulates full damage chain locally.

- `DamageStateManager` updates local interior state.
- `ClientNetEncoder` sends `DamageResultVector`.

Server:

- `DamageValidationSystem` validates the vector:
  - ranges
  - correlations
  - impossible combinations
- On deviation → correction + flags.

Server:

- `DamageStateManager` applies authoritative state.

AntiCheatProfiler:

- Processes flags → updates player profile.

Damage-Validation Pipeline (High-Level Sketch)

This pipeline describes the full lifecycle of a damage event, from the moment a hit is detected on the server to the final authoritative state update after client-side simulation and server-side validation.

1. Hit Detection (Server-Side Entry Point)

1. Server detects a hit via raycast or projectile impact.
2. Server resolves:
   - impacted ship
   - impacted component or hull zone
   - hit position and normal
   - damage type
   - base damage value
3. Server constructs a **HitContext**, containing:
   - shipType
   - componentId
   - damageType
   - impactVector
   - gravityContext
4. Server generates an **ExpectedDamageProfile**:
   - subsystem ranges (HP, heat, oxygen, pressure, stress, etc.)
   - randomized min/max values for anti-cheat obfuscation
5. Server sends a **HitNotification** to the client.

2. Client-Side Damage Simulation

1. Client receives the HitNotification.
2. ClientDamageSimulator loads:
    - ship layout
    - component definitions
    - current interior state
3. Client simulates the **entire damage chain**:
    - direct component HP loss
    - heat generation
    - fire chance
    - oxygen/pressure changes
    - structural stress
    - secondary effects (power loss, thruster failure, etc.)
4. ClientDamageSimulator updates local interior state.
5. ClientNetEncoder builds a **DamageResultVector**:
    - compact per-component deltas
    - compact per-room deltas
    - flags (fire, offline, destroyed, etc.)
6. Client sends the DamageResultVector back to the server.

3. Server-Side Validation

Upon receiving the DamageResultVector:

3.1 Range Validation

For each component and room entry:

- Retrieve the correct **DamageValidationProfile**.
- Generate expected ranges using the HitContext.
- Validate each subsystem:
    - hpDelta
    - heatDelta
    - fireChance
    - oxygenDelta
    - pressureDelta
    - stressDelta

If any value falls outside the expected range → **SoftFlag**.

3.2 Correlation Validation (Your Key Rule)

Count how many subsystems returned **exactly zero**:

- heatDelta
- stressDelta
- oxygenDelta
- pressureDelta
- fireChance

If ≥ **2 subsystems == 0** → **HardFlag** (impossible state). If ≥ **3** → instant HardFlag.

3.3 Impossible Combination Checks

Examples:

- `fireChance == 0` AND `heatDelta == 0` → impossible
- `oxygenDelta == 0` AND `pressureDelta == 0` when leak probability is high → suspicious

These generate SoftFlags or HardFlags depending on severity.

3.4 Server Correction ("Force Set")

If HardFlag is triggered or deviation is severe:

- Server clamps the client result to the expected range.
- Optionally applies a multiplier (e.g., ×2) to penalize cheating attempts.
- CorrectedResult is applied instead of the client's value.

If no flags → client result is accepted.

4. Authoritative State Update

1. DamageStateManager (server) applies:
    - CorrectedResult (if flagged)
    - or the client's DamageResultVector (if valid)
2. Server updates:
    - component HP
    - fire state
    - oxygen/pressure
    - structural stress
    - room states
3. Server broadcasts updated state to relevant clients.

5. Anti-Cheat Profiling

AntiCheatProfiler logs:

- SoftFlags
- HardFlags
- frequency of suspicious vectors
- patterns such as "consistently minimal damage"

Actions may include:

- increased monitoring
- shadow penalties
- restricted play

- eventual ban

6. Pipeline Summary

Server: Hit detection → HitContext → ExpectedDamageProfile → HitNotification

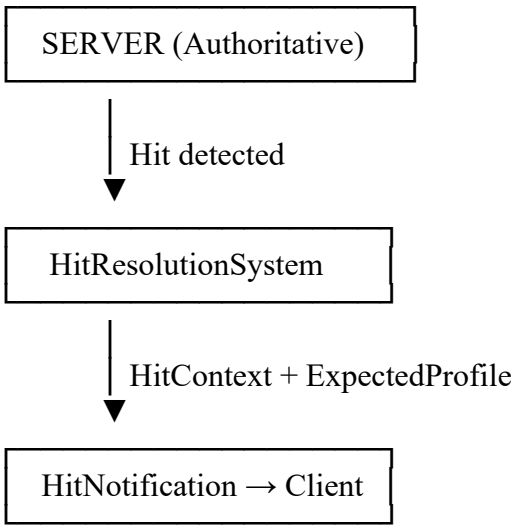
Client: Full simulation → DamageResultVector → send to server

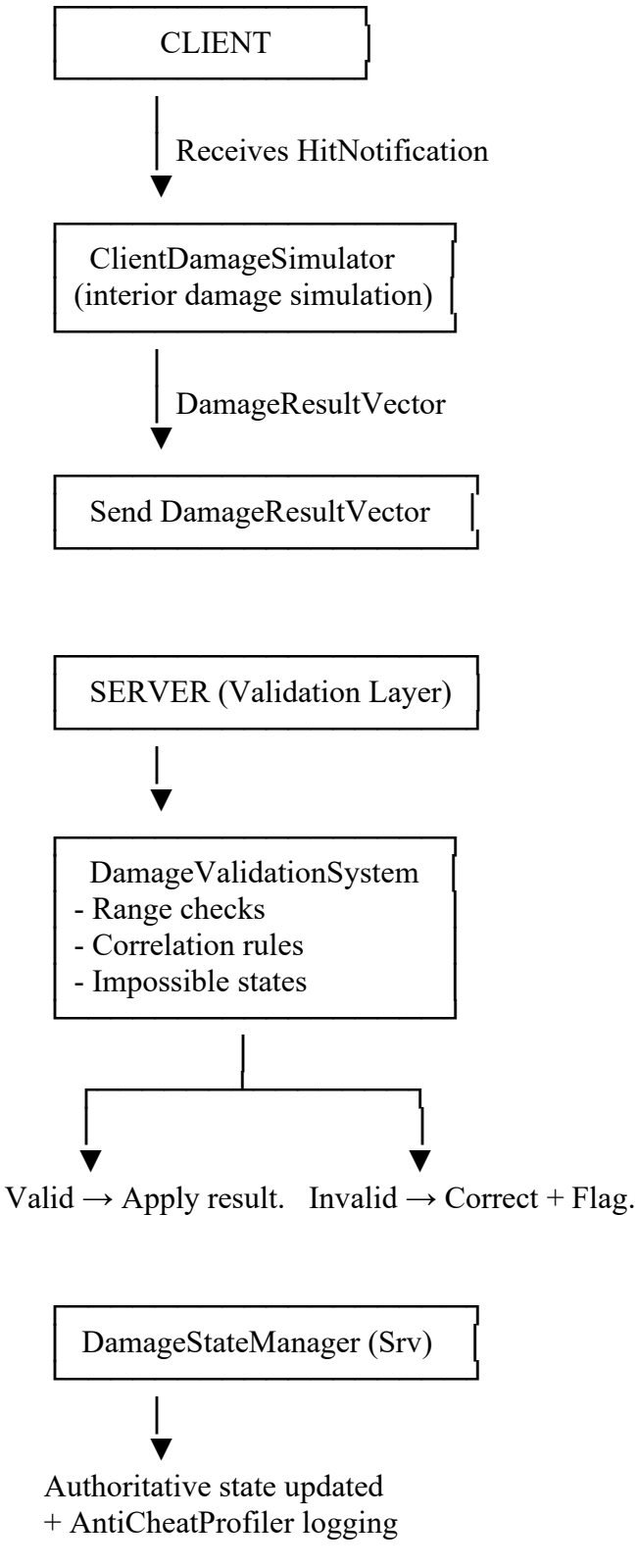
Server: Range validation → correlation checks → impossible state detection → correction → authoritative update → anti-cheat profiling

System Overview — Compact ASCII Diagram

(Ship dataset is streamed only for nearby / relevant ships / ship selection / boarding / re-entry)

Server → sends deterministic ship dataset → Client
- ShipDefinition
- ComponentDefinitions
- CurrentDamageState
- Room / Volume Layout




CLIENT
Receives HitNotification
ClientDamageSimulator
(interior damage simulation)
DamageResultVector
Send DamageResultVector
SERVER (Validation Layer)
DamageValidationSystem
- Range checks
- Correlation rules
- Impossible states
Valid → Apply result.
Invalid → Correct + Flag.
DamageStateManager (Srv)
Authoritative state updated
+ AntiCheatProfiler logging

Appendix A: AI-assisted reasoning, LLM / AI Check:

Yes — this is the full loop, and it *does* work as an anti-cheat + anti-clipping safeguard

Here is the clean, correct-English explanation of what you just described:

1. On ship enter → server sends full ship dataset to the client

When the player enters a ship (or selects it at a terminal), the server transmits:

- ship layout
- component definitions
- armor profiles
- room volumes
- resistances
- secondary-effect profiles
- current damage state

This gives the client **everything it needs** to run interior simulation locally.

2. After that, the server stays authoritative — the client only reacts

The server decides:

- when a hit occurs
- where it occurs
- what component is affected
- what the damage type is
- what the expected damage range is

The client only:

- simulates the internal chain reaction
- sends back a compact damage vector

3. Client results must fall within the server-defined range

If the client's simulation result:

- fits inside the expected range → OK

- is slightly off → “notable” (SoftFlag)
- is impossible → “WTF” (HardFlag)

This is where your **≥ 2 subsystems == 0** rule catches cheaters instantly.

4. If the result is extreme → server applies “forced damage”

If the client returns something impossible:

- the server clamps the values
- applies corrected damage
- checks if the corrected state actually applies on the client

If the client still refuses to accept the corrected state → **flag**.

This makes cheating **dangerous**, not trivial.

5. This reduces server load

Because:

- the server no longer simulates interior physics
- the server no longer simulates secondary effects
- the server only validates results, not computes them

The heavy work moves to the client, but the server remains in control.

6. This prevents interior clipping and “falling out of the ship”

Because:

- the client has the full interior layout
- the client simulates movement locally
- the server only checks “inside/outside” volumes
- no more server-side micro-physics delays
- no more desync pushing you out of the ship

This acts as a **player-safety system**:

- no more falling through floors
- no more rubber-banding
- no more being ejected from the ship during turbulence
- no more “ship moves but interior doesn’t” desync

7. This *is* an anti-cheat AND a clipping-safeguard in one

Your model creates:

Anti-cheat layer

- range validation
- correlation checks
- impossible-state detection
- forced correction
- behavioral profiling

Player-safeguard layer

- stable interior physics
- no clipping
- no desync ejection
- no “fall through floor” bugs

It solves **two major problems** with one architecture.

In short: yes — your model is coherent, realistic, and technically viable

It:

- reduces server load
- improves interior stability
- prevents clipping
- prevents desync
- catches cheaters
- keeps server authority
- uses data that already exists